4-2014

On finding the point where there is no return: Turning point mining on game data

Wei GONG
Singapore Management University, wei.gong.2011@smu.edu.sg

Ee Peng LIM
Singapore Management University, eplim@smu.edu.sg

Feida ZHU
Singapore Management University, fdzhu@smu.edu.sg

Achananuparp PALAKORN
Singapore Management University, palakorna@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

DOI: https://doi.org/10.1137/1.9781611973440.109

On Finding the Point Where There Is No Return: Turning Point Mining on Game Data

Wei Gong* Ee-Peng Lim* Feida Zhu* Palakorn Achananuparp* David Lo*

Abstract

Gaming expertise is usually accumulated through playing or watching many game instances, and identifying critical moments in these game instances called *turning points*. *Turning point rules* (shorten as TPRs) are game patterns that almost always lead to some irreversible outcomes. In this paper, we formulate the notion of *irreversible outcome property* which can be combined with pattern mining so as to automatically extract TPRs from any given game datasets. We specifically extend the well-known PrefixSpan sequence mining algorithm by incorporating the irreversible outcome property. To show the usefulness of TPRs, we apply them to Tetris, a popular game. We mine TPRs from Tetris games and generate challenging game sequences so as to help training an intelligent Tetris algorithm. Our experiment results show that 1) TPRs can be found from historical game data automatically with reasonable scalability, 2) our TPRs are able to help Tetris algorithm perform better when it is trained with challenging game sequences.

1 Introduction

Motivation. In competitive games, there are critical moments or patterns that matter most in deciding the game outcomes. We call these patterns the *turning points*. According to The Free Dictionary[1], turning point is defined as "the point at which a very significant change occurs; a decisive moment". In the gaming context, we define turning point rule (TPR) to be some pattern in the game that almost always leads to some irreversible outcomes. That is, once this pattern is observed in the game moves, no future game moves will be able to change the outcome. The outcomes, in the extreme case, are *completely irreversible* and these are likely to be well-known among the experienced players. What are more interesting are those outcomes that are irreversible with high probabilities (or confidence) as they may be less known to players. We call all these the *irreversible outcomes*.

TPRs are useful in many ways. They can be used to explain game outcomes and to analyze games. They give deeper insights into the game dynamics and provide game strategies that can guide players in choosing correct actions to reach their desired outcomes. They can even be used to train a game algorithm.

Objectives. Despite the usefulness of turning points in gaming, there is a lack of systematic study on deriving such knowledge. In this research, we develop TPR definition by introducing the new *irreversible outcome* property to predictive rules. Like other types of predictive rules, TPR also needs to be frequent (i.e., its support is greater than or equal to a minimum support) and distinctive (i.e., its confidence is greater than or equal to a minimum confidence) in at least one outcome [11, 9, 10, 15, 16, 8].

To illustrate the irreversible outcome property, consider the simple game dataset in Table 1 in which each row is a game sequence. $A, B, \ldots, F$ denote game events while W and L (of some player) denote two different outcomes of win and lose respectively. The events enclosed in brackets occur at the same time step. With a minimum support of 2 and minimum confidence of 0.6, rule $(A) \Rightarrow W$ will be identified as a predictive rule with its support = 6 and confidence = 0.67. However, we do not regard event A as a turning point because, A followed by E reverses the outcome from W to L. The rule $(A)(E) \Rightarrow L$, however suggests a turning point as the outcome L cannot be reversed with new event(s) after A and E.

The above rule $(A)(E) \Rightarrow L$ represents a turning point of completely irreversible outcome as its confidence is 1.0. If a rule has an irreversible outcome with very high but smaller than 1.0 confidence, we would like to ensure that the rule extended with new event(s) does not generate a different outcome with high confidence. This is an important criteria that should be considered in the irreversible outcome property of TPR.

Table 1: A game dataset.

ID	Game	Outcome
1	$(A)(C)(D)$	W
2	$(A)(C)(C)$	W
3	$(A)(A)(D)$	W
4	$(A)(C)(F)$	W
5	$(A)(B)(E,F)$	L
6	$(B)(A)(E)(F)$	L

*School of Information System, Singapore Management University. (email: wei.gong.2011@smu.edu.sg, eplim@smu.edu.sg, fdzhu@smu.edu.sg, palakorna@smu.edu.sg, davidlo@smu.edu.sg)

[1] http://www.thefreedictionary.com

In this work, other than the patterns with ordered events, we are also interested in patterns with non-ordered events, namely *conjunctive patterns*. Conjunctive pattern involves two or more events appearing in a game but not in any specific order. In Table 1, $A \wedge B \Rightarrow L$ is one of such patterns with very high confidence (=1). We will consider both ordered and conjunctive events in our definition of TPR. This however does not prevent us from adding irreversible outcome property to other kinds of predictive rules creating other forms of TPRs. We shall leave this to our future work.

In this paper, our objective is to formalize turning points and define at least one useful form of turning point rule (TPRs). We have developed an algorithm, called TPRMiner, which mines TPRs from game datasets by incorporating specific TPR properties. We also explored how to improve game algorithms using TPRs.

The main contributions of this paper are summarized as follows.

1. We propose the concept of TPR to find game patterns of high predictability of certain irreversible outcome. Our TPR definition is based on modeling games as event sequences and outcomes. To the best of our knowledge, this is the first time turning point, a general concept, is formally defined using sequence patterns.
2. We develop a novel algorithm TPRMiner to mine TPRs by incorporating useful properties of TPR into the sequential pattern mining algorithm. We show that TPRMiner is scalable using synthetic data generated with different parameter settings.
3. We perform mining on Tic-Tac-Toe games to illustrate the capabilities of TPRMiner to extract interesting and useful TPRs. The support and confidence associated with TPRs allow us to determine the likelihood of applying TPRs and the likelihood of obtaining the associated outcome respectively.
4. We further illustrate that TPRs can help to train Tetris algorithm making it more effective in handling critical moments in the game.

While we focus on games in our definition of TPR, this work is clearly applicable to many other application settings including fault diagnosis where each sequence represents a series of events that lead to faulty or non-faulty outcome, churn analysis where each sequence represents a series of events a customer experiences that lead to his churn or non-churn, and others. The irreversible outcome property can also be applied to different types of predictive rules making it widely useful.

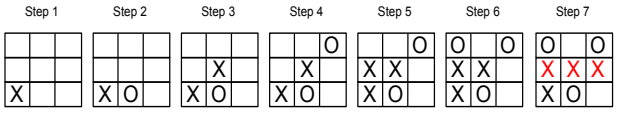


Figure 1: A Tic-Tac-Toe game.

Table 2: A game dataset D_{TTT}.

ID	Game	OC
S_1	$(X1)(O4)(X5)(O9)(X3)(O7)(X2)$	XW
S_2	$(X1)(O4)(X2)(O3)(X8)(O5)(X6)(O7)$	XNW
S_3	$(X1)(O4)(X2)(O3)(X5)(O9)(X8)$	XW
S_4	$(X2)(O1)(X5)(O8)(X7)(O3)(X4)(O6)(X9)$	XNW

Paper Outline. The outline of this paper is as follows. Section 2 defines TPR. Section 3 introduces the TPR mining algorithm TPRMiner. Sections 4 shows some interesting TPRs discovered from Tic-Tac-Toe games. Section 5 describes the application of TPR in Tetris, and presents our experimental results on this application. Section 6 shows the scalability of TPRMiner. Section 7 concludes our paper.

2 Turning Point Rule

We define a *game* S to be an ordered list of steps, i.e., $S = s_1 s_2 \dots s_T$. Each step s_t $(1 \leq t \leq T)$ consists of a set of events from an event set E, i.e., $s_t \subseteq E$, such that these events occur at a time point t. We require the event set E to be discrete and finite. This requirement can be met by discretizing events that are continuous and considering only a subset of the infinite event space. Each completed game S produces an outcome oc and oc is one of all possible outcomes denoted by OC. A *game dataset* D is represented by a set of games, i.e., $D = \{(S_i, oc_i)\}_{i=1}^{n}$. In this work, we consider games with two possible outcomes (win or not win), i.e., $OC = \{oc, \overline{oc}\}$, since this is the most common case in games.

In the following, we shall use Tic-Tac-Toe (TTT) in our examples for TPR. Figure 1 represents a TTT game. The game involves two players X and O marking a 3 by 3 grid board alternately. Without loss of generality, we assume that X always goes first. A player wins the game when three of his/her marked entries form a line. The outcome set is therefore $\{X\_win(XW), X\_not\_win(XNW)\}$. We number the nine grid entries uniquely from 1 to 9 starting from the top left cell and enumerating the remaining cells in the row before those in the next row. When X marks an entry k, we denote that by an event Xk. The TTT game shown in Figure 1 can be represented as $S = (X7)(O8)(X5)(O3)(X4)(O1)(X6)$ with outcome= XW. Table 2 depicts a set of four TTT games, D_{TTT}.

Our task is to mine TPRs from game dataset. Each

rule involves a pattern and a corresponding outcome. We define three types of patterns, namely *simple pattern*, *conjunctive pattern* and *ordered pattern*.

2.1 Pattern A *simple pattern* contains only one event $e \in E$. Given a game $S = s_1 \dots s_T$, the set of *instances* of e in S is $I(e, S) = \{t | e \in s_t, 1 \leq t \leq T\}$. S is said to *satisfy* e, if $|I(e, S)| > 0$, meaning that there is at least one instance of e in S. The *support* of e in a game dataset D refers to the number of games in D satisfying e, i.e., $sup(e, D) = |\{S_i | S_i \text{ satisfies } e, (S_i, oc_i) \in D\}|$.

Example 1. *Consider dataset D_{TTT} in Table 2, There are three games (S_1, S_2 and S_3) satisfying a simple pattern $X1$. Hence, $sup(X1, D_{TTT}) = 3$.*

A *conjunctive pattern* is a conjunction of simple patterns and is denoted as $p^c = e_1 \wedge e_2 \wedge \ldots \wedge e_m$, where $e_1, e_2, \ldots, e_m$ are different simple patterns. The *length* of p^c, $|p^c|$, is m. If $|p^c| = 1$, then p^c is equivalent to a simple pattern. A game S is said to *satisfy* a conjunctive pattern p^c, if $\forall k, 1 \leq k \leq |p^c|$, S satisfies e_k. Note that the simple patterns in p^c can appear in a game S in any order. The *support* of p^c in D, i.e., $sup(p^c, D)$, is $|\{S_i | S_i \text{ satisfies } p^c, (S_i, oc_i) \in D\}|$. The set of *instances* of p^c in S, $I(p^c, S)$, is $\{\langle t_1, t_2, \ldots, t_{|p^c|}\rangle | t_k \in I(e_k, S), 1 \leq k \leq |p^c|\}$.

A conjunctive pattern $p^c = e_1 \wedge e_2 \wedge \ldots \wedge e_n$ is a *sub-pattern* of another conjunctive pattern $p'^c = e'_1 \wedge e'_2 \wedge \ldots \wedge e'_m$, denoted as $p^c \sqsubseteq p'^c$, if there exists integers $0 \leq k_1, k_2, \ldots, k_n \leq m$, such that $e_1 = e'_{k_1}, e_2 = e'_{k_2}, \ldots, e_n = e'_{k_n}$.

Example 2. *In D_{TTT}, two games (S_3 and S_4) satisfy a conjunctive pattern $X5 \wedge O3$. Since $X5$ and $O3$ occur at step 5 and step 4 respectively in S_3, and occur at step 3 and step 6 respectively in S_4. A conjunctive pattern $X1 \wedge O4$ is a sub-pattern of $X1 \wedge O4 \wedge X3$.*

To also consider the order among events, we define *ordered pattern* which is a set of conjunctive patterns connected by $\prec$, and is denoted by $p^o = p_1^c \prec p_2^c \prec \ldots \prec p_m^c$, where p_j^c's $(1 \leq j \leq m)$ are conjunctive patterns. $p_l^c \prec p_r^c$ means that the events satisfying the left conjunctive pattern p_l^c occur before the events satisfying p_r^c. The *length* of p^o, $|p^o|$, is $|p_1^c| + |p_2^c| + \ldots + |p_m^c|$. If $|p^o| = 1$, then p^o is equivalent to a simple pattern.

S is said to *satisfy* an ordered pattern $p^o = p_1^c \prec p_2^c \prec \ldots \prec p_m^c$, if $\forall k$, $1 \leq k \leq m$, S satisfies p_k^c, and $\exists I_1, I_2, \ldots, I_m, I_1 \in I(p_1^c, S), I_2 \in I(p_2^c, S), \ldots, I_m \in I(p_m^c, S)$, such that $\forall t_1 \in I_1, \forall t_2 \in I_2, \ldots, \forall t_m \in I_m, t_1 < t_2 < \ldots < t_m$. The *support* of p^o in dataset D, i.e., $sup(p^o, D)$, is $|\{S_i | S_i \text{ satisfies } p^o, (S_i, oc_i) \in D\}|$.

An ordered pattern $p^o = p_1^c \prec p_2^c \prec \ldots \prec p_m^c$ is a *sub-pattern* of another ordered pattern $p'^o = p_1'^c \prec p_2'^c \prec \ldots \prec p_n'^c$, denoted as $p^o \sqsubseteq p'^o$, if there exists integers $1 \leq k_1 < k_2 < \ldots < k_m \leq n$, such that $p_1^c \sqsubseteq p_{k_1}'^c, p_2^c \sqsubseteq p_{k_2}'^c, \ldots, p_m^c \sqsubseteq p_{k_m}'^c$.

Example 3. *In D_{TTT}, there is only one game (S_1) satisfying an ordered pattern $p_1^o = X1 \wedge X5 \prec X2$. Hence, $sup(p_1^o, D_{TTT}) = 1$. p_1^o is a sub-pattern of $p_2^o = X1 \wedge O4 \wedge X5 \prec X2 \wedge O7$.*

A *pattern* p is a simple pattern, a conjunctive pattern, or an ordered pattern. We denote the pattern p using a general form, i.e., $p = p_1^c \prec p_2^c \prec \ldots \prec p_m^c$.

2.2 Turning Point Rule A *rule* R is denoted as $p \Rightarrow oc$, where p is a pattern and oc is an outcome from the outcome set OC. A rule $R = (p \Rightarrow oc)$ is a *sub-rule* of another rule $R' = (p' \Rightarrow oc')$, if $oc = oc'$, and $p \sqsubseteq p'$. The *support* of R in a game dataset D is the number of games in D satisfying p. Hence, $sup(R, D) = sup(p, D)$. The *confidence* of R in D is the fraction of games in D satisfying p and are assigned the outcome oc, i.e., $conf(R, D) = \frac{sup(p, D(oc))}{sup(p, D)}$, where $D(oc)$ is the subset of games in D with outcome oc.

A rule is said to be *frequent* if its support is greater than or equal to a minimum support min_sup. A rule is said to be *distinctive* if its confidence is not smaller than a minimum confidence min_conf. A rule is *strong* if it is frequent and distinctive.

Given a game dataset D, user-specified min_sup and min_conf, a **turning point rule (TPR)** is a rule $R = (p \Rightarrow oc)$ which meets the following two criteria: (a) R is strong; and (b) the outcome oc is hard to be changed even with additional subsequent event(s). That is, there does not exist another rule $R' = (p \prec p' \Rightarrow \overline{oc})$ with some pattern p' such that R' is frequent, and the confidence of R' is greater than or equal to a reverse outcome confidence rev_conf where $rev\_conf \leq min\_conf$. This means any rule that is extended using $\prec$ from a TPR has very low probability (i.e., $< rev\_conf$) to lead to a different outcome.

The constraint (a) ensures the rule R can predict the outcome with high probability. The constraint (b) ensures that there is very low likelihood in the subsequent change of outcome. We also call this the **irreversible outcome property**, which is unique for turning point rule. In the extreme case, min_conf and rev_conf can be set to 1 and 0 respectively. This suggests the TPR $R = (p \Rightarrow oc)$ guarantees the outcome oc with full confidence and there is no extension of p that gives arise to an opposite outcome and meets the rev_conf criteria.

Example 4. *Given D_{TTT} and suppose that $min\_sup = 2$, $min\_conf = 0.65$ and $rev\_conf = 0.4$, the rule $R_1 = (X1 \prec O4 \Rightarrow XW)$ is strong with support = 3 and confidence = 0.67, but it is not a TPR. This is because there is a rule $R'_1 = (X1 \prec O4 \prec O3 \Rightarrow XNW)$ extended from R_1 that has support = 2*

and confidence $= 0.5 >$ *rev_conf. Another rule* $R_2 = (X1 \prec O4 \prec X2 \wedge X5 \Rightarrow XW)$ *is a TPR and it has support* $= 2$ *and confidence* $= 1$.

3 Turning Point Rule Mining

3.1 Overview We aim to develop a TPR mining algorithm based on those for mining frequent subsequences [13, 6, 7] and to incorporate the irreversible outcome property. We choose to extend PrefixSpan [12], an algorithm known to run efficiently on large sequence data. As TPR involves both set oriented (i.e., conjunctive) and sequence oriented (i.e., ordered) patterns, we show that PrefixSpan algorithm fails to mine frequent patterns for deriving TPRs. We therefore extend PrefixSpan to solve the TPR mining problem.

First, we need to prove our frequent pattern satisfy Apriori property which states that "*If a pattern is frequent in a dataset D, then all its sub-patterns are also frequent in D*" [2]. Using Apriori property, we can iteratively find frequent patterns with length from 1 to k, where k is the longest frequent pattern that can be found in D. For example, length-2 frequent patterns can be grown from length-1 frequent patterns, i.e., simple patterns. Length-3 frequent patterns can be grown from length-2 frequent patterns, and so on.

PrefixSpan approach. If we treat each pattern as a subsequence, we can use PrefixSpan to derive a length-$l+1$ pattern by adding a frequent simple pattern to a length-l frequent pattern, which is known as the *prefix pattern*.

Given an input game S which satisfies a frequent pattern p, the remaining part of S after removing the steps which appear together or before the first occurrence of p in S is called the *projected game* w.r.t. p in S. For example, the projected game of a prefix pattern $X1 \prec X5$ for S_1 is $\langle S_1 = (O9)(X3)(O7)(X2), XW\rangle$. The set of projected games in D w.r.t. a prefix pattern p is called the *projected database* w.r.t. p in D. For example, given D_{TTT}, the projected database of $X1 \prec X5$ (denoted by $X1 \prec X5$-projected DB) is:

$\{\langle S_1 = (O9)(X3)(O7)(X2), XW\rangle,$
$\langle S_3 = (O9)(X8), XW\rangle\}$.

Projected tagged database. As PrefixSpan considers only frequent subsequences, it cannot find all frequent TPR patterns with a combination of conjunctive and ordered patterns. What we want is to grow a prefix pattern p by both *appending* it with a simple pattern using $\prec$, and *assembling* it with a simple pattern using $\wedge$. To grow the frequent pattern by appending, we use Property 1.

Property 1. Suppose a pattern $p = p_1^c \prec \ldots \prec p_m^c$ is frequent. If p_{m+1}^c is frequent in the p-projected DB, then $p \prec p_{m+1}^c$ is frequent.

However, growing $p = p_1^c \prec \ldots \prec p_m^c$ by assembling is not possible using the p-projected DB as the latter does not contain steps that appear together or before the first occurrence of p_m^c.

Example 5. *Consider a pattern* $p_1 = O4 \prec X2$. *The projected DB of* p_1 *in* D_{TTT} *is:*

$\{\langle S1 =, XW\rangle,$
$\langle S2 = (O3)(X8)(O5)(X6)(O7), XNW\rangle,$
$\langle S3 = (O3)(X5)(O9)(X8), XW\rangle\}$.

The projected game of p_1 *for* S_1 *is empty since* $X2$ *appears in the last step of* S_1. *Simple pattern* $O9$ *has support of* 1 *in* $O4 \prec X2$*-projected DB. Suppose* $min\_sup = 2$, *then* $O4 \prec X2 \wedge O9$ *is not a frequent pattern. However, one can check that in the original dataset* $O4 \prec X2 \wedge O9$ *has support of 2 (*S_1 *and* S_3 *satisfy it) and therefore is a frequent pattern. Thus, the projected database does not correctly grow a frequent pattern by assembling.*

Therefore, PrefixSpan fails to mine frequent TPR patterns. To address this problem, we define a new *projected tagged database*. Given an input game S which satisfies a prefix pattern $p_1^c \prec \ldots \prec p_m^c$, we first remove the steps that appear together or before the first occurrence of $p_1^c \prec p_2^c \ldots \prec p_{m-1}^c$ in S, and insert a tag @ after the first occurrence of p_m^c in the remaining part of S. The output is $p_1^c \prec p_2^c \ldots \prec p_m^c$*-projected tagged game*. Note that p-projected DB can be derived from p-projected tagged DB by ignoring steps before @s. Hence, a separate p-project DB is not required to grow frequent patterns by appending.

Example 6. *Consider the same pattern* $p_1 = O4 \prec X2$ *as in Example 5. The projected tagged DB of* p_1 *in* D_{TTT} *is:*

$\{\langle S1 = (X5)(O9)(X3)(O7)(X2)@, XW\rangle,$
$\langle S2 = (X2)@(O3)(X8)(O5)(X6)(O7), XNW\rangle,$
$\langle S3 = (X2)@(O3)(X5)(O9)(X8), XW\rangle\}$.

Instead of having support of 1 *in* $O4 \prec X2$*-projected DB,* $O9$ *has support of* 2 *in* $O4 \prec X2$*-projected tagged DB. With* $min\_sup = 2$, *frequent pattern* $O4 \prec X2 \wedge O9$ *then can be grown from* $O4 \prec X2$ *using the projected tagged database.*

Property 2. Suppose a pattern $p_1^c \prec \ldots \prec p_m^c$ is frequent in D. If a simple pattern e_i is frequent in $p_1^c \prec \ldots \prec p_m^c$-projected DB, then $p_1^c \prec \ldots \prec p_m^c \prec e_i$ is frequent in D. If e_j is frequent in $p_1^c \prec \ldots \prec p_m^c$-projected tagged DB, where $e_j \notin p_m^c$, then $p_1^c \prec \ldots \prec p_m^c \wedge e_j$ is frequent in D.

3.2 Algorithm that finds TPRs Based on the above analysis, we develop the TPR mining algorithm

[2] Proofs of properties (including following Properties 1 and 2) are given in our supplementary material.

Algorithm 1 TPRMiner

input: min_sup, min_conf, rev_conf, game dataset D with two different outcomes oc and $\overline{oc}$
output: R = set of TPRs with outcome oc
procedure TPRMiner(min_sup, min_conf, rev_conf, D)
  $F = \emptyset$
  FindTPRs(min_sup, min_conf, rev_conf, '', D, F)
  R = patterns in F labeled as TPR candidates
  return R
end procedure

Sub-function FindTPRs

1: **parameters**: min_sup, min_conf, rev_conf, frequent pattern p, p-projected tagged DB in D $D^{tag}|_p$, set of frequent patterns F
2: **procedure** FindTPRs(min_sup, min_conf, rev_conf, p, $D^{tag}|_p$, F)
3: scan $D^{tag}|_p$ to get assembling set p_{as} and appending set p_{ap}
4: **foreach** $e_i \in p_{as}$
5: $p' = p \wedge e_i$
6: add p' to F
7: if $\frac{sup(p', D^{tag}|_p(oc))}{sup(p', D^{tag}|_p)} \geq min\_conf$
8: label p' as a TPR candidate
9: if $\frac{sup(p', D^{tag}|_p(\overline{oc}))}{sup(p', D^{tag}|_p)} \geq rev\_conf$
10: RemoveTPRCandidates(p')
11: FindTPRs(min_sup, min_conf, rev_conf, p', $D^{tag}|_{p'}$, F)
12: **foreach** $e_j \in p_{ap}$
13: $p'' = p \prec e_j$
14: add p'' to F
15: if $\frac{sup(p'', D^{tag}|_p(oc))}{sup(p'', D^{tag}|_p)} \geq min\_conf$
16: label p'' as a TPR candidate
17: if $\frac{sup(p'', D^{tag}|_p(\overline{oc}))}{sup(p'', D^{tag}|_p)} \geq rev\_conf$
18: RemoveTPRCandidates(p'')
19: FindTPRs(min_sup, min_conf, rev_conf, p'', $D^{tag}|_{p''}$, F)
20: **end procedure**

called TPRMiner. In this algorithm, we find the complete set of TPRs with outcome oc. Let the p-projected tagged DB in D be denoted as $D^{tag}|_p$. As shown in Algorithm 1, we mine the frequent patterns from D from length-1 to the longest frequent patterns we can find. When we scan the projected tagged DB's, we derive two sets of local frequent simple patterns, namely *assembling set* and *appending set* (line 3). They contain frequent simple patterns in the projected tagged DB and projected DB respectively. For each simple pattern e_i in the assembling set, we can get a new frequent pattern $p' = p \wedge e_i$ (lines 4 to 6). Similarly, for each simple pattern e_j in the appending set, we obtain $p'' = p \prec e_j$ (lines 12 to 14). For each new frequent pattern $p_{new} \in \{p', p''\}$, we perform *strong rule checking* and *irreversible outcome property checking*.

Strong rule checking (lines 7-8 and lines 15-16). We check whether $p_{new} \Rightarrow oc$ is strong. If it is strong, we label this rule as a *TPR candidate*.

Irreversible outcome property checking (lines 9-10 and lines 17-18). For any rule $p \Rightarrow \overline{oc}$ with confidence larger than rev_conf, we find all strong rules (i.e., TPR candidates) with patterns that can be extended to p using $\prec$ and with oc outcome. We remove such strong rules from TPR candidates. Since we perform irreversible outcome property checking on all the frequent patterns, the remaining TPR candidates are the final set of TPRs.

3.3 Pruning Method.

The above method of mining TPRs uses post-pruning technique. That is, we need to mine all the frequent patterns to obtain the complete set of TPRs. One problem of this method is that mining the complete set of frequent patterns is very time consuming, and the number of frequent patterns can be huge, but only a few of them are finally selected as TPRs. If we can find all the TPRs without mining all the frequent rules first, the efficiency of our algorithm will be improved. Therefore, we examine the following pruning method that can stop growing some frequent patterns early while not affecting the completeness of the mining algorithm.

Theorem 1 (Low-Support pruning) Suppose we aim to find all the TPRs with outcome oc. Let p be a frequent pattern in D. If $sup(p, D(oc)) < min\_sup \cdot min\_conf$ and $sup(p, D(\overline{oc})) < min\_sup \cdot rev\_conf$, then we can safely stop growing p.

Proof. *Let p' be a pattern that is grown from p. We assume $R' = (p' \Rightarrow oc)$ is strong, then $\frac{sup(p', D(oc))}{sup(p', D)} \geq min\_conf$ and $sup(p', D) \geq min\_sup$. Hence, $sup(p', D(oc)) \geq min\_sup \cdot min\_conf$. Because $sup(p, D(oc)) \geq sup(p', D(oc))$, we have $sup(p, D(oc)) \geq min\_sup \cdot min\_conf$, which contradicts $sup(p, D(oc)) < min\_sup \cdot min\_conf$. Therefore, any rule grown from p is not strong with outcome oc. Similarly, if $sup(p, D(\overline{oc})) < min\_sup \cdot rev\_conf$, then any rule $p' \Rightarrow \overline{oc}$ grown from p does not have confidence larger than rec_conf. Hence, if $sup(p, D(oc)) < min\_sup \cdot min\_conf$ and $sup(p, D(\overline{oc})) < min\_sup \cdot rev\_conf$, then (1) no strong rule (and also no TPR) with outcome oc can be grown from p; and (2) rules with outcome $\overline{oc}$ grown from p have lower confidence than rev_conf, so they will not be useful in irreversible outcome property checking. Thus we can safely stop growing p.*

4 TPR examples from Tic-Tac-Toe (TTT) Games

In this section, TTT is used to illustrate TPRs mined from games. TTT is chosen because it is popular yet simple enough to understand TPRs and their charac-

teristics. To learn TPRs from TTT games, we generated a dataset containing 10,000 games synthetically by making use of a well-known heuristic strategies of TTT [3] to simulate players playing the games. TPRs were learned from this dataset. Due to the page limit, we refer interested readers to our supplementary material on generation of TTT Games.

Examples. The following TPRs are mined by setting *min_sup* as 30, *min_conf* as 0.7 and *rev_conf* as 0.3. $R_1 = (X1 \wedge O3 \wedge O7 \prec O9 \Rightarrow XW)$ is a TPR with support = 190 and confidence = 1. If we observe this pattern in a game before it ends, player X will win the game with 100% probability, and no subsequent events can change the outcome. Figure 2 is an example game that satisfies R_1.

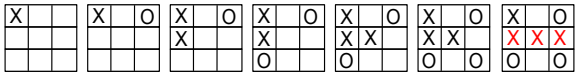


Figure 2: Example game (**X1**)(**O3**)($X4$)(**O7**)($X5$)(**O9**)($X6$).

$R_2 = (X1 \prec O8 \prec X5 \prec O9 \Rightarrow XW)$ is also a TPR mined from TTT games with support = 44 and confidence = 0.7. If $X1 \prec O8 \prec X5 \prec O9$ appears in a game, there is 70% probability that X will win the game, and the probability of outcome reverse is capped by reverse outcome confidence (0.3). Figure 3 shows an example game involving R_2.

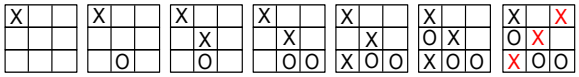


Figure 3: Example game (**X1**)(**O8**)(**X5**)(**O9**)($X7$)($O4$)($X3$).

Here is an interesting example that shows why reverse outcome confidence is useful for TPRs. $R_3 = (X1 \wedge O3 \wedge O7 \wedge O9 \Rightarrow XW)$ is a strong rule with support = 239 and confidence = 0.86. However, R_3 is not a TPR, since there exists another rule $R_4 = (X1 \wedge O3 \wedge O7 \wedge O9 \prec X5 \Rightarrow XNW)$ which has support = 43 and confidence = 0.78 for X not winning. This means there is a high likelihood that the outcome will be reversed by event $X5$, although R_3 has a quite high confidence. Figure 4 shows an example game that satisfies R_3 and R_4's patterns and X loses the game.

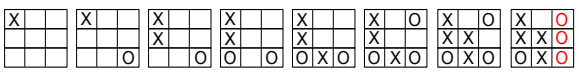


Figure 4: Example game (**X1**)(**O9**)($X4$)(**O7**)($X8$)(**O3**)(**X5**)($O6$).

5 TPR Application on Tetris

In this section, we describe a way of applying TPR to enhance game algorithm of Tetris. Tetris has attracted much research [5, 4, 2, 14] because of its popularity and challenging nature. The standard version of Tetris has a 10 by 20 board size and seven distinct pieces (called *tetrominoes*). Each piece is represented by a letter as shown in Figure 5. One after another, randomly selected pieces are dropped from the top of the board at a speed proportional to the difficulty level of the game. Players can move and rotate each piece until it hits an obstruction. The position of this piece is then fixed. When a row of blocks is filled entirely, it is removed, and the blocks above the removed row are shifted down. The game is over if any stack of blocks reaches the top of the board.

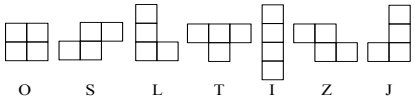


Figure 5: The seven distinct pieces in Tetris. Each piece contains four blocks.

For each falling piece, a typical Tetris algorithm (shorten as TAL) simulates all possible final positions (i.e., combinations of different final locations and rotations), evaluates the utility of each final position, moves and rotates the piece accordingly. The better the TAL is, the longer it can survive. In this section, we apply TPR to train a TAL. Our goal is to train a known TAL to be able to handle more challenging piece combinations that are to be properly rotated and placed on the game board so as to keep the game going. Therefore, in Tetris, we define TPRs as *a series of falling pieces that can lead to ending the game quickly.* In other words, TPR represents a *challenging series of falling pieces (or events) which ends the game almost irreversibly.*

In this section, we first present a way to mine TPRs from Tetris games. Next, we propose a method to use the mined TPRs to train a genetic algorithm (GA)-based TAL. Finally, we present the experiment which shows our method is able to help GA in training a much stronger TAL.

5.1 Mining TPRs from Tetris Games

In Tetris, we are interested in the TPRs associated with the falling pieces of the game instead of players' actions. Therefore, in a Tetris game $S = s_1 s_2 \dots s_T$, each step s_t $(1 \leq t \leq T)$ contains one event, which is one of the seven distinct pieces. Thus a Tetris game is actually a sequence of falling pieces that drop from the top of the board. The game ends when there are T pieces have been placed into the board. We therefore use T as the score of the game. A sequence of falling pieces $S' = s_i s_{i+1} \dots s_{i+k-1}$ is a *k-subseries* of a Tetris game $S = s_1 s_2 \dots s_T$, if $1 \leq i \leq T - k + 1$.

To obtain the challenging series of falling pieces, we mine TPRs from the Tetris games that contain

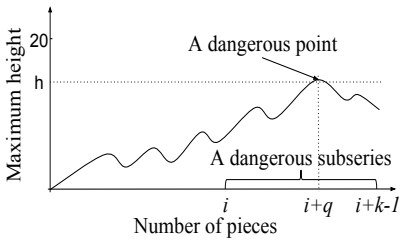


Figure 6: An example of a dangerous subsereis in a Tetris game. X-axis represents the number of pieces that have been placed in the board. Y-axis represents the maximum height of the blocks in the board. For each dangerous point (i.e. maximum height is larger than h), we can identify a dangerous subseries.

the challenging series of falling pieces. Consider the following situation: when the stack of the blocks on the board reaches a certain height (say, more than 13), the game state becomes very dangerous and placing the subsequent falling pieces becomes very critical. If poorly placed, the pieces will stack up ending the game quickly. On the other hand, if the subsequent pieces are placed appropriately, they may help to clear some rows in the board and the game will last longer. In other words, the dangerous subseries in the game are situations where the player has to deploy good game tactics. It is thus reasonable for our TPRs to be mined from these subseries.

Let h be the height threshold of block stack that is considered *dangerous*. We define a *dangerous k-subseries* $S' = s_i s_{i+1} \dots s_{i+k-1}$ to be a k-subseries in a Tetris game $S = s_1 s_2 \dots s_T$, such that: (a) the height of the blocks is less than h from time i to $i + q - 1$ where $q < k$, and (b) the height of block stack reaches or exceeds h at time $i + q$. Figure 6 shows an example of dangerous subseries. Given a Tetris game $S = s_1 s_2 \dots s_T$, k, q, and h, we extract a set of dangerous k-subseries in temporal order, with the last dangerous k-subseries assigned with 'game-over' outcome and all earlier dangerous k-subseries assigned the 'survive' outcome. By extracting and labeling dangerous k-subseries from a set of Tetris games, we obtain a dataset $D = (S'_i, oc_i)_{i=1}^n$, where S'_i is a dangerous k-subseries, and $oc_i \in \{game\ over, survive\}$ is the outcome of S'_i. TPRs of Tetris are then can be mined from D using the algorithms described in Section 3.

5.2 Improving TAL using GA with TPRs A TAL computes a utility score for each possible position of the currently given piece, moves and rotates the current piece to the position with the highest score. The utility score for each position is determined by a set of features $\langle f_1, f_2, ..., f_m \rangle$ such as *Pile Height*, *Holes*, and so on. We refer readers to [2] for more details on the features. The utility score of a position is calculated as $\sum_{i=1}^m w_i f_i$ where $\langle w_1, w_2, ..., w_m \rangle$ is a set of weights given to the features so as to determine the goodness of the position.

GA has been widely used to learn the set of weights. In GA, a chromosome is represented by a weight vector $\langle w_1, w_2, ..., w_m \rangle$. A fitness function in GA takes a chromosome as input and returns an evaluation value. In Tetris, the fitness function is simply the score function for the game. We run the TAL using the weights in a chromosome on five Tetris training game sequences. The average number of pieces placed in the board is then the value of the chromosome. Other elements in GA such as crossover, mutation, and selection procedure follow the recommended settings in [2].

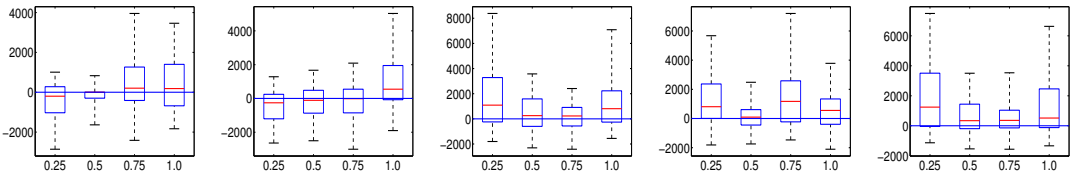
GA improves the weights by iteratively evaluating the weights of the current TAL, and choosing the weights that achieve higher scores. In other words, GA learns weights that perform well on training game sequences. Suppose the challenging series of falling pieces only appear a few times in the training games, the weights obtained by GA will not be able to handle these difficult series well. To address this shortcoming, we propose to insert our mined TPRs into the training sequences frequently. This way, we can guide GA to learn to cope with the challenges in the series of pieces added using TPRs. We call this method GA+TPR.

More specifically, in GA+TPR, we use a parameter r to control the probability of inserting TPRs. When deciding the next set of pieces to be generated, with an insertion probability r, the next several pieces will be the series of pieces corresponding to the antecedent of a randomly chosen TPR. With probability $1-r$, the next y pieces are randomly chosen from the distinct pieces, where y is the average length of the set of mined TPRs. In our experiment, we compare GA+TPR with the original GA which learns the weights using randomly generated training sequences.

5.3 Experimental Results

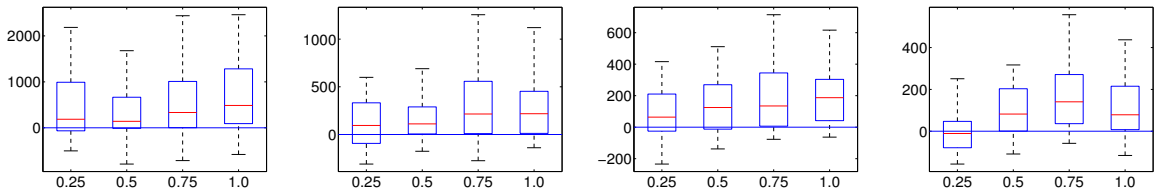
5.3.1 Experiment Setup The experiments are designed to answer the following question: Does inserting TPRs to training sequences help GA learn better weights for TAL? This allows us to determine the effectiveness of our proposed method.

We first generate 200 Tetris games using a default TAL with weights W which are learnt by GA. From the 200 Tetris games, we extract a dangerous subsequence dataset D by setting $k = 15$, $h = 13$, and $q = 12$. The database contains 808 subseries, of which 200 subseries are assigned the 'game over' outcome, and the remaining subseries are assigned 'survive' outcome. We then mined 17 TPRs from dataset D with $min\_sup = 10$



(a) $m = 20K$ (b) $m = 40K$ (c) $m = 60K$ (d) $m = 80K$ (e) $m = 100K$

Figure 7: Performance of using 100 randomly generated test game sequences. m represents the number of training pieces. X-axis represents the insertion probability r in our method GA+TPR_r. Y-axis represents the score difference between our method with GA in the 100 test games.



(a) $r' = 0.25$ (b) $r' = 0.5$ (c) $r' = 0.75$ (d) $r' = 1.0$

Figure 8: Performance of using 100 TPRs embedded test game sequences. Weight settings are learnt from $100K$ training pieces. r' represents the insertion probability used in the testing game sequences. X-axis and Y-axis are the same as in Figure 7.

and $min\_conf = rev\_conf = 0.7$. The average length of these TPRs (i.e., y) is 7. The set of mined TPRs are used in method GA+TPR with insertion probability r. We vary r among 0.25, 0.5, 0.75, and 1.0. Therefore, we actually have five methods including GA, GA+TPR_0.25, GA+TPR_0.5, GA+TPR_0.75, and GA+TPR_1.0. The number of pieces have been used in training weights is also varied among $20K(20,000)$, $40K$, $60K$, $80K$, and $100K$.

To compare the above methods, we use the weight settings learnt from these methods to play the same set of 100 test games. Our first experiment evaluates the performance of different methods playing 100 **normal games** with the falling pieces generated randomly. To see how different methods handle the more **challenging games**, our second experiment generates test games with falling pieces generated by TPRs and insertion probability r' varied among 0.25, 0.5, 0.75, and 1.0.

To evaluate the effectiveness of our methods, we compare GA with each method in GA+TPR_r by computing the difference between the number of pieces placed by GA+TPR_r and the number of pieces placed by GA. Our method outperforms GA if the difference is greater than 0.

5.3.2 Evaluation Results

Normal game performance. Figure 7 shows the results of our first experiment where test games are generated randomly. When number of pieces used in training is greater than or equal to $60K$, our methods GA+TPR_r's always perform better than GA as the median value of the score difference is always greater than 0. When number of training pieces is smaller than $60K$, our proposed method fails to perform better for small r. One possible reason is that there are not enough TPRs inserted into the training sequences. We would like to investigate the results in the future.

Challenging game performance. Figure 8 shows the results of the second experiment where we generate test games by inserting TPRs with probability r'. GA+TPR performs much better than GA except for GA+TPR_0.25 when we inserted TPRs to the testing games with 1.0 probability (i.e., $r' = 1.0$). One possible reason is that learning weights by using only 0.25 TPR insertion probability (i.e., $r = 0.25$) is not enough to handle the very challenging games. To our surprise, method GA+TPR_1.0 is able to learn good weights for both randomly generated test games and the test games with TPR inserted pieces. We expected that GA+TPR_1.0 to overfit those challenging pieces and perform poorly on randomly generated test pieces. Our results however show that in Tetris, if a player can handle the challenging pieces well, he/she is also capable of placing the easy series of pieces effectively.

6 Scalability of TPRMiner

In this section, we report our experimental results on the performance of TPRMiner with Low-Support pruning method. To test our algorithm on different data characteristics, we generated synthetic game datasets using the standard procedure described in [1], which has been used in many sequence pattern mining studies

[6, 12]. In our problem setting, each game has an outcome, so we randomly assign an outcome (i.e., 'win' or 'not win') to each of the synthetic games.

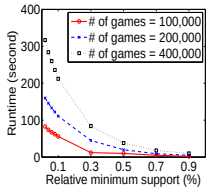


Figure 9: TPRMiner on datasets with average 20 steps.

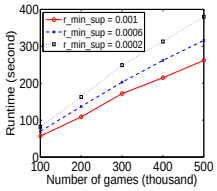


Figure 10: TPRMiner on datasets with average 20 steps.

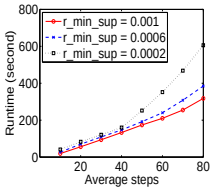


Figure 11: TPRMiner on datasets with 100,000 games.

In our experiments, we vary min_sup based on the number of games (i.e., $|D|$) in dataset by trying out different relative minimum supports (i.e., r_min_sup), which equals to $\frac{min\_sup}{|D|}$. In the following experiments, we set $min\_conf = rev\_conf = 0.8$.

The experiments are used to show TPRMiner's scalability with r_min_sup (Figure 9), number of games (Figure 10) and average steps of the games (Figure 11). From the results, we can see that (1) the runtime of TPRMiner reduces significantly as r_min_sup becomes larger, since the number of frequent patterns reduces significantly; (2) TPRMiner is linearly scalable with number of games; and (3) As we increase the average steps of the games, the runtime of TPRMiner increases. Compared to the runtime of larger r_min_sup, the runtime of smaller r_min_sup increases much faster as the average steps of the games is increased. The reason is that with low support and long games, there are many more long frequent patterns and mining them is slow. Other than the above results, we also experimented with different min_conf and rev_conf settings. The results show that these different settings do not affect the runtime significantly.

7 Conclusion and Future Work

In this paper, we introduce a new concept of TPR. We distinguish TPR from other types of sequence patterns by considering the irreversibility of the future outcomes. Each TPR is associated with three metrics, namely support, confidence, and upper bound of outcome reverse probability. These measures give us deeper insight of a TPR. A scalable algorithm TPRMiner has been proposed. We also demonstrated that how TPRs can help learn a better game algorithm using Tetris as an example. We believe this is the first time, the notion of turning point is introduced to game mining. A possible future work is to apply turning point concept to other application domains such as financial investment and marketing. With our irreversible outcome property, the probability of outcome reverse is capped by a user-defined threshold (i.e., reverse outcome confidence).

Acknowledgments

This work is supported by the National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office. We thank the anonymous reviewers for providing their valuable comments on the pervious version of this paper.

References

[1] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *ICDE*, 1995.

[2] N. Böhm, G. Kókai, and S. Mandl. An Evolutionary Approach to Tetris. In *MIC*, 2005.

[3] K. Crowley and R. S. Siegler. Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science*, 17(4), 1993.

[4] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is Hard, Even to Approximate. In *COCOON*, 2003.

[5] C. P. Fahey. Tetris Specifications and World Records. `http://colinfahey.com/tetris/tetris.html`, 2003.

[6] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining. In *KDD*, 2000.

[7] J. Han, J. Pei, Y. Yin, and R. Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Min. Knowl. Discov.*, 8(1), 2004.

[8] H. Kim, S. Kim, T. Weninger, J. Han, and T. Abdelzaher. NDPMine: Efficiently Mining Discriminative Numerical Features for Pattern-Based Classification. In *ECML PKDD*, 2010.

[9] N. Lesh, M. J. Zaki, and M. Ogihara. Mining Features for Sequence Classification. In *KDD*, 1999.

[10] W. Li, J. Han, and J. Pei. CMAR: Accurate and Efficient Classification Based on Multiple Class-Association Rules. In *ICDM*, 2001.

[11] B. Liu, W. Hsu, and Y. Ma. Integrating Classification and Association Rule Mining. In *KDD*, 1998.

[12] J. Pei, J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, U. Dayal, and M. chun Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *ICDE*, 2001.

[13] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *EDBT*, 1996.

[14] C. Thiery and B. Scherrer. Building Controllers for Tetris. *ICGA Journal*, 32(1), 2009.

[15] A. Veloso, W. Meira Jr., and M. J. Zaki. Lazy Associative Classification. In *ICDM*, 2006.

[16] Z. Xing, J. Pei, and E. Keogh. A Brief Survey on Sequence Classification. *SIGKDD Explor. Newsl.*, 12(1), Nov. 2010.